entwickler.de

windows .developer

APPS MODERNISIEREN

FIT FÜR DIE ZUKUNFT

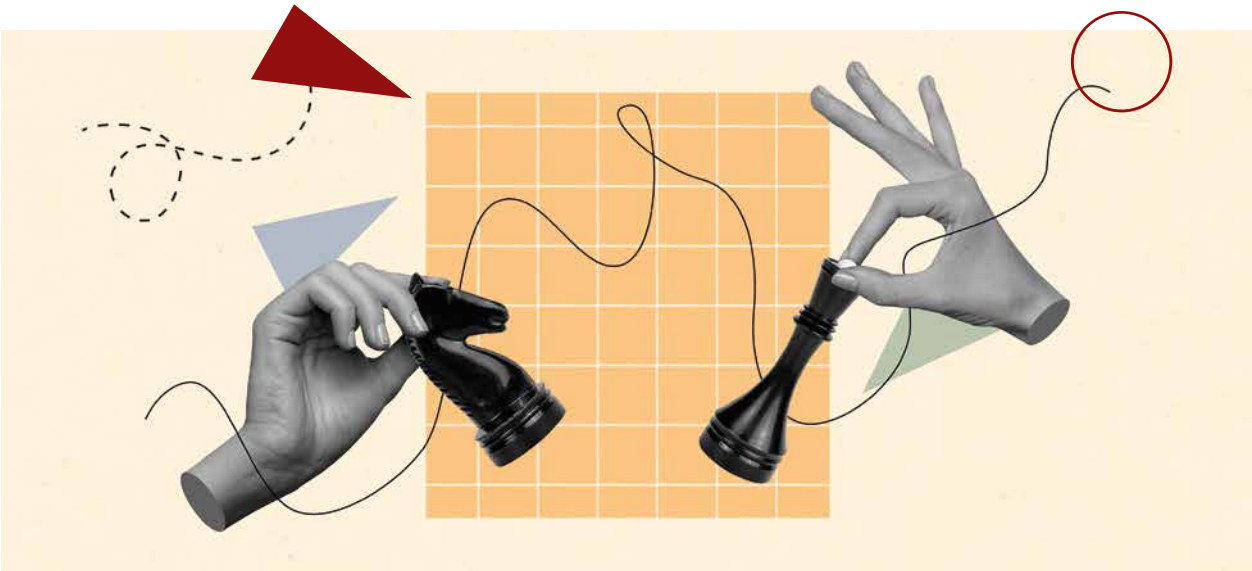
Sonderdruck für fecher

LEGACY APP MODERNIZATION
Wie fange ich an?

ANWENDUNGS-MODERNISIERUNG
Methodik und Stolperfallen

WISEJ.NET
Browsermigration für Windows-Apps

© viagraphix.net/shutterstock.com,
© S&S Media



© Roman Samborskyi/shutterstock.com

Methodik und Stolperfallen der Anwendungsmodernisierung

Legacy auf die Höhe der Zeit gebracht

Viele Unternehmen stehen heute vor der Frage, wie sie ihre Legacy-Geschäftsanwendungen modernisieren und an die aktuellen und zukünftigen Anforderungen anpassen wollen. Doch die richtige Methodik zur Modernisierung zu finden, ist nicht einfach. Neben den technischen Aspekten geht es hauptsächlich auch um strategische, organisatorische und wirtschaftliche Fragen. Dieser Artikel richtet sich an alle, die sich mit der Modernisierung von Businessanwendungen beschäftigen oder vor dieser Aufgabe stehen.

von Günter Hofmann

Ob für den Einsatz im eigenen Unternehmen entwickelt oder als kommerzielle Software für den Wiederverkauf – heute erfolgreiche Geschäftsanwendungen haben eine jahrelange, oft jahrzehntelange Historie. Sie sind über lange Zeit gewachsen und tragen eine Fülle an Funktionalität und Daten mit sich. Aber nicht alles, was vor 20 Jahren eine gute Plattform- oder Architekturentscheidung war, scheint in unserer heutigen IT-Welt noch sinnvoll. Daraus resultieren die häufigsten Gründe für die Notwendigkeit einer Anwendungsmodernisierung:

- Teilweise werden die Technologien, mit denen die Software seinerzeit entwickelt wurde, vom jeweiligen Hersteller nicht mehr weiterentwickelt. So lassen sich etwa klassische Visual-Basic-Anwendungen heute nur noch eingeschränkt auf aktuellen Windows-Versionen einsetzen und viele moderne Schnittstellen und Sicherheitsverfahren stehen nicht zur Verfügung.
- Es wird immer schwieriger, Entwickler:innen für ältere Programmiersprachen zu finden. Die Anzahl der Gupta-Spezialist:innen auf dem Markt beispielsweise ist ohnehin überschaubar, zudem stehen viele der verbleibenden Expert:innen kurz vor der Rente. Nachwuchs, der die IT mit modernen Umgebungen kennengelernt hat, lässt sich für die alten Technologien kaum mehr begeistern.
- Generell sind heute oft Architekturen notwendig, die von den Legacy-Plattformen nicht unterstützt wer-

den. Ob Plattformunabhängigkeit, installationsfreie Nutzung oder Cloud-Technologie – historische Anwendungen, die in einer Fat-Client-Architektur ohne Schichtentrennung realisiert sind, bieten nicht die erwartete Flexibilität.

- In den letzten Jahren hat sich eine Reihe von neuen Lizenzmodellen etabliert, die aktuelle Plattformen erfordern. Dank Abrechnungsverfahren wie Pay-per-Use, Subscription oder Metered müssen die Unternehmen nicht mehr hohe Summen fest in Softwarelizenzen investieren, während Softwarehäuser mit stabileren Einnahmen rechnen können.
- Auch einzelne Anwender:innen profitieren von modernen Plattformen, wenn es darum geht, ihre Software nicht mehr nur auf dem eigenen Desktop, sondern auch auf mobilen Geräten oder am Heim-PC per Internetverbindung einzusetzen. Oft ist eine Webanwendung die ideale Lösung für diese grundlegende Änderung der Usability.
- Das klassische, am Windows-Desktop der Neunzigerjahre orientierte Erscheinungsbild von Legacy-Anwendungen wirkt altbacken und spricht weder die bestehenden Nutzer:innen an, die damit arbeiten müssen, noch ist es verkaufsfördernd für die Neukundengewinnung.

Die entscheidenden Gründe für eine Modernisierung sind also betriebswirtschaftlicher und nicht primär technischer Natur. Entsprechend sollte auch die Lösung auf betriebswirtschaftlicher Grundlage erfolgen. In diesem Sinne kann eine Modernisierung dann als gelungen betrachtet werden, wenn sie den vollständigen technischen Umbau mit dem geringsten Aufwand erreicht – also in kürzester Zeit und zu niedrigsten Kosten. Um Zielkonflikte zwischen technischen und kaufmännischen Entscheidungsträger:innen von vornherein zu vermeiden, ist hier eine klare Festlegung wichtig.

Wie schlecht ist Legacy wirklich?

Ein ähnlicher Zielkonflikt zeigt sich auch in der Bewertung von Legacy-Software. Ganz allgemein bezeichnet man mit diesem Begriff zunächst wertfrei eine Software, die vor vielen Jahren entwickelt wurde und noch in der Nutzung ist. Legacy-Software kann sowohl Vorteile als auch Nachteile haben, wobei die vorrangige Sichtweise je nach Kulturkreis variiert.

Im deutschsprachigen Raum wird Legacy-Software oft negativ konnotiert und vorrangig die technische Schuld mit Faktoren wie Sicherheit, Leistung und Kompatibilität betrachtet: Legacy-Software kann anfälliger für Sicherheitslücken sein als neuere Software, insbesondere wenn die enthaltenen Technologien nicht mehr weiterentwickelt werden. Legacy-Software kann oft nicht mit den steigenden Anforderungen an Leistung und Skalierbarkeit Schritt halten. Legacy-Software ist häufig nicht mit neueren Technologien oder Systemen kompatibel. Das kann zu Problemen bei der Integration in heutige IT-Umgebungen führen.

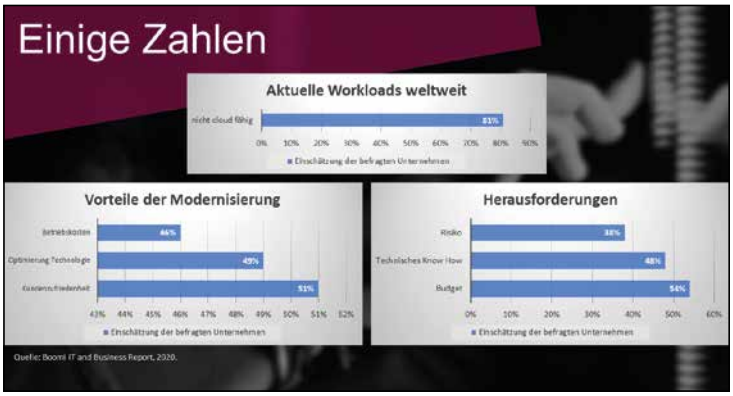


Abb. 1: Ergebnisse des Boomi IT and Business Report 2020

Im angloamerikanischen Raum hingegen tendiert man eher zur Betrachtung der positiven wirtschaftlichen Aspekte mit den Faktoren Funktionalität, Daten und Kosten: Legacy-Software enthält sämtliche Funktionalität, die über all die Jahre aufgebaut wurde und die Geschäftsprozesse optimal unterstützt. Legacy-Software enthält auch die über die Jahre gesammelten Daten, die für die Unternehmen von großem Wert sind und für Analyse, Entscheidungsfindung und operative Zwecke zur Verfügung stehen. Nicht zuletzt ist Legacy-Software grundsätzlich kostengünstiger als die Entwicklung neuer Software, da sie bereits vorhanden ist und funktioniert.

Beide Betrachtungsweisen haben ihre Berechtigung, jeder Kulturkreis kann vom anderen etwas lernen. Deshalb gilt es bei der Planung eines Modernisierungsprojekts, dieses sehr sorgfältig darauf auszulegen, die Vorteile der Legacy-Software zu erhalten und die Nachteile mit möglichst geringem Aufwand an Zeit und Ressourcen zu eliminieren. Überspitzt gesagt: Wenn das Ergebnis eine technisch perfekte Lösung ist, die fünf Jahre zu spät auf den Markt kommt und die Hälfte der Funktionalität vermissen lässt, wird das Modernisierungsprojekt kein wirtschaftlicher Erfolg werden.

Zielsetzungen

Das wird in den Unternehmen ähnlich verstanden: In einem IT and Business Report von Boomi aus dem Jahr 2020 [1] gaben die befragten Unternehmen an, den größten Vorteil der Modernisierung in der Kundenzufriedenheit zu sehen (51 Prozent), vor der zweitgenannten Optimierung der Technologie (49 Prozent) und dem weiteren wichtigen Faktor Betriebskosten (46 Prozent). Als größte Herausforderungen wurden genannt: Budget (54 Prozent), technisches Know-how (48 Prozent) und Risiko (38 Prozent). Gleichzeitig scheint die Modernisierung weiter hinterherzuhinken als allgemein angenommen. So schätzen die befragten Unternehmen, dass immerhin 81 Prozent des weltweiten Anwendungs-Workloads über nicht Cloud-fähige Systeme und Lösungen abgewickelt werden (**Abb. 1**).

Daraus lassen sich konkrete Zielsetzungen für die einzelnen Bereiche einer Anwendungsmodernisierung ableiten (**Abb. 2**). Über allem steht der Wunsch der Anwender:innen nach gleichbleibender Funktionalität.

Abb. 2: Zielsetzungen bei der Anwendungsmodernisierung

Für Softwarehäuser ist dieser sogar doppelt wichtig, definiert hier die Funktionalität nicht nur die Akzeptanz der Bestandskunden für die modernisierte Lösung, sondern vor allem die Größe der Zielgruppe, die für das zukünftige Neukundengeschäft adressiert werden kann. Einen Rückschritt beim Funktionsumfang, und sei er nur vorübergehend, gilt es daher unbedingt zu vermeiden.

Sofern die bestehende Lösung auf einer älteren Programmiersprache basiert, die in den modernen Umgebungen nicht mehr gut unterstützt wird, gilt es, diese durch eine zukunftssichere Sprache zu ersetzen. In der Microsoft-.NET-Welt betrifft das etwa VB 6 oder Gupta. Als Zielsprache dürfte nach der effektiven Abkündigung von VB.NET durch Microsoft mittlerweile C# gesetzt sein.

Die Herkunft aus der Zeit bis zur Jahrtausendwende ist älteren Anwendungen meist deutlich anzusehen. Eine gelungene Modernisierung des User Interface ist mehr als ein reines Facelift. Sie umfasst auch die Themen Benutzerfreundlichkeit und -produktivität (Usability), Einsatz auf verschiedenen Gerätetypen (Responsiveness) und Barrierefreiheit (Accessibility). Damit trägt sie viel zur wahrgenommenen Qualität der modernisierten Software bei.

Ein Refactoring der Architektur von Client/Server hin zu einer Browseranwendung ist ebenfalls ein wichtiges Ziel in fast jedem Modernisierungsprojekt. Selbst wenn zunächst kein Einsatz im Web geplant ist, sind die Vorteile umfangreich: Es ist keine Softwareinstallation mehr auf dem Client erforderlich und die Anwendung für jeden Nutzenden stets aktuell nutzbar mit Zugriff auf die neuesten Daten. Zudem können nahezu beliebige Endgeräte eingesetzt werden und es ist die Webfähigkeit, die die meisten Deployment-Modelle aus der Cloud erst möglich macht.

Allerdings darf die Nutzung nicht nur per Web und Browser erfolgen können. Ein plattformunabhängiges Deployment, das neben der Live-Ausführung der Anwendung aus Inter- oder Intranet auch den lokalen oder sogar hybriden Einsatz ermöglicht, ist im Sinne der größtmöglichen Flexibilität für den zukünftigen Einsatz sinnvoll. Dabei sollten neben den Desktopsystemen Windows und Linux auch Mobile-Plattformen wie Android und iOS unbedingt unterstützt werden.

Bei aller Technologie muss unbedingt auch das Team berücksichtigt werden, das am Ende die modernisierte Anwendung beherrschen und weiterhin fortentwickeln und pflegen muss. Hilfreich ist dafür, wenn die Software trotz aller Veränderungen durch die Modernisierung in ihren bisherigen Grundstrukturen weiterhin erkennbar bleibt. Dennoch müssen neue Entwicklungen möglich sein, ohne die technische Historie der Anwendung im Detail zu kennen. Alte wie zukünftige Teammitglieder sollen sich mit der modernisierten Software also wohlfühlen können.

Ein oft übersehener, letzter Faktor ist die Zeitschiene. Hier geht es nicht nur darum, dass mit der für die Modernisierung verwendeten Zeit die Kosten steigen, obwohl das auch ein wichtiger Aspekt ist. Vor allem aber ist eine im Einsatz befindliche Software ein bewegtes Ziel für jede Anpassung – schließlich kann die Weiterentwicklung in den wenigsten Fällen für Jahre angehalten werden, bis die Modernisierung abgeschlossen ist. Alle vorgenannten Aspekte in einer realistischen Zeitschiene zu berücksichtigen, wird im Zweifelsfall nur mit einem toolbasierten und weitgehend automatisierten Modernisierungsansatz gelingen können.

Grün oder braun?

Überhaupt muss man sich bei einem solchen Projekt überlegen, auf welcher Basis es aufsetzen soll. Wer eine Anwendung neu konzipiert, wünscht sich oft, damit jede Entscheidung völlig frei und von Grund auf neu treffen zu können – man spricht dann von einem Greenfield-Ansatz. Baut man das Projekt hingegen auf bereits bestehenden Strukturen und Prozessen auf, ist das der sogenannte Brownfield-Ansatz.

Bezogen auf die Modernisierung von Windows-Desktop-Anwendungen lässt sich sagen, dass Greenfield heute meist einer vollständigen Neuentwicklung in der Zielsprache C# entsprechen dürfte, für den Cloud-Einsatz aufgeteilt in möglichst unabhängige Microservices und mit einem User Interface in HTML und JavaScript und weiteren Bibliotheken von Drittherstellern. Was technisch spannend klingt, bringt jedoch einige Herausforderungen mit sich: Sie ist ein vollständiger Neuentwurf mit entsprechend hoher Komplexität und dem damit verbundenen Aufwand für die Umsetzung. Daraus resultiert eine lange Zeit bis zur Marktreife, während der eine Parallelentwicklung von alter und neuer Anwendung unvermeidbar ist. Obendrein wird für die Neuentwicklung regelmäßig ein komplett neues Softwareteam mit entsprechenden technischen Qualifikationen benötigt, das sich in die Geschäftsprozesse der Anwendung allerdings erst aufwendig einarbeiten muss. Wenn die Anwendung schließlich fertig ist, wird bei jeder einzelnen Kundeninstallation eine Datenmigration

aus der Altanwendung erforderlich. Und ob die Bestandskunden die aus ihrer Sicht neue und anders aufgebaute Anwendung akzeptieren werden, bleibt fraglich.

Eine Brownfield-Modernisierung, wie sie im Unternehmen des Autors durchgeführt wird, setzt hingegen auf dem vorhandenen Code auf und unterzieht diesen einem gründlichen Refactoring. Das UI, das in der Legacy-Anwendung Teil einer monolithischen Gesamtapplikation ist, wird dabei unter Beibehaltung der enthaltenen Logik technisch separiert und die ganze Migration erfolgt so weit wie möglich werkzeugbasiert. Nur für einzelne kritische Teile wird eine Neuentwicklung erforderlich, die wahlweise selbst übernommen oder outgesourct werden kann. So bleiben Geschäftslogik und Datenschicht erhalten und lediglich die Programmiersprache muss ausgetauscht werden, sofern bislang nicht bereits C# zum Einsatz kam. Die Erfahrung des vorhandenen Teams kann genutzt werden und es muss kein zweites Team mit zusätzlicher Qualifikation aufgebaut werden. Durch den Einsatz eines geeigneten Frameworks lässt sich die hohe Komplexität des bisherigen Desktop-UI auch in der neuen Webanwendung abbilden. Die kürzere Zeit bis zur Marktreife reduziert die notwendige Parallelentwicklung, der Erhalt der bisherigen Geschäftsprozesse und Bedienschritte sichern eine hohe Akzeptanz bei den Bestandskunden.

Abb. 3: Stolperfallen auf dem Weg ins Web

Standortbestimmung auf dem Weg ins Web

Gehen wir im Folgenden also davon aus, dass eine vorhandene Altanwendung nicht vollständig neu geschrieben, sondern per Modernisierung, idealerweise toolbasiert, auf den Stand einer modernen Browserapplikation unter .NET gebracht werden soll. Je nach der bisher eingesetzten Programmiersprache, UI-Technologie und Architektur, ergeben sich unterschiedliche Bewertungen bezüglich der Zukunftsfähigkeit, der Plattformunabhängigkeit, der Qualifikation des Softwareteams sowie der möglichen Toolunterstützung auf dem Weg ins Web (**Abb. 3**). Bei der Programmiersprache ergibt sich folgendes Bild:

- Wer bereits auf C# setzt, verfügt über eine sehr gute Basis hinsichtlich der Zukunftsfähigkeit. Mit .NET Core, also den Versionen 7 und 8, besteht dann auch die Möglichkeiten der Plattformunabhängigkeit. Das Softwareteam kennt sich bereits bestens mit .NET aus und kann mit C# eine zukünftige Webanwendung problemlos pflegen und weiterentwickeln. Ein toolbasiertes Web-Enabling ist mit den entsprechenden Werkzeugen jederzeit möglich.
- Basiert die Altanwendung auf VB.NET, gibt es schon gewisse Problemstellen – .NET 7 und 8 werden voraussichtlich nicht mehr voll unterstützt, entsprechend wäre die Zukunftsfähigkeit nicht mehr gegeben. Eine Plattformunabhängigkeit kann ebenfalls nicht erreicht werden. Das Team kennt sich mit .NET bereits gut aus und kann C# vermutlich rasch lernen. Am Ende bleibt die Möglichkeit, toolbasiert von VB.NET nach C# zu migrieren und dann ist die Ausgangslage wieder optimal. Als Stolperfalle muss man allerdings im Kopf behalten, dass C# nicht alle VB.NET-Konstrukte beherrscht, hier ist unter Umständen bei der Migration zusätzlich ein manuelles Refactoring vorzusehen.
- Mit einer Legacy-Sprache wie VB 6 oder Gupta ist Zukunftsfähigkeit heute grundsätzlich nicht mehr gegeben. Auch die Plattformunabhängigkeit liegt in weiter Ferne. Dem Entwicklungsteam hilft seine Kenntnis der Legacy-Programmiersprache in Zukunft nicht mehr weiter. Hier tut also Weiterbildung oder Ergänzung um neue Mitglieder mit entsprechender Expertise Not. Mit spezialisierten Werkzeugen ist eine toolbasierte Migration aber durchaus möglich. Diese könnte grundsätzlich erst einmal zu VB.NET erfolgen, was insbesondere von VB 6 kommend naheläge. Aus den vorgenannten Gründen ist der direkten Migration nach C# dennoch unbedingt der Vorzug zu geben. Eine entscheidende Stolperfalle liegt in der Qualität der Altanwendung. Besteht in dieser bereits eine hohe technische Schuld, wird diese bei der Migration nicht kleiner werden. Dieser Weg eignet sich daher nur für eine aktiv gepflegte Legacy-Anwendung oder mit einem vorgeschalteten begrenzten Refactoring.

Bezüglich der eingesetzten Technologie für das User Interface lässt sich festhalten:

- Wo als UI-Technologie heute WinForms eingesetzt wird, kann die Zukunft kurzfristig als sicher gelten. Allerdings ist nach wie vor unklar, wie lange Microsoft das Konzept noch weiterverfolgen wird und eine Plattformunabhängigkeit ist nicht gegeben. Das Team kann mit dem vorhandenen Know-how weiterarbeiten und ist auch für die Webumgebung unter .NET vorbereitet. Ein Web-Enabling kann mit den entsprechenden Werkzeugen automatisiert vorgenommen werden.

- Basiert die Benutzeroberfläche auf Legacy-Plattformen wie VB 6 oder Gupta, ist die Zukunftsfähigkeit erst einmal nicht gesichert – maximal nach einer (automatisierten) Migration nach WinForms. Die Plattformunabhängigkeit ist nicht gegeben und das Team hat mit seinem Wissen über die Legacy-Plattform auf Dauer keine Zukunft. Eine toolbasierte Migration von Legacy nach WinForms ist möglich, ebenso von WinForms nach Web. Die entscheidende Stolperfalle an dieser Stelle ist eine betriebswirtschaftliche: Vielfach wird angenommen, dass eine Webanwendung nur durch aufwendige Neuentwicklung mit Technologien wie Angular/React/JavaScript möglich wird, obwohl ein toolbasiertes Web-Enabling mittlerweile zu den etablierten Verfahren zählt. Eine weitere Stolperfalle liegt in der Versuchung, Legacy-Anwendungen, die heute noch auf VB 6 oder Gupta basieren, lediglich nach WinForms zu migrieren. Selbstverständlich ist das möglich, bedeutet aber einen Wechsel von einer Client/Server-Technologie zu einer anderen mit ebenfalls begrenzter Zukunftssicherheit. Hier empfiehlt sich in jedem Fall ein vollständiges Web-Enabling. Solange kein echter Webbetrieb erforderlich ist, kann die Anwendung zunächst dennoch im Client/Server-Betrieb verwendet werden, etwa mit einem Hybrid- oder Desktopansatz oder als PWA (Progressive Web App).

Für die Anwendungsmodernisierung genutzte Werkzeuge

Neben den Standardwerkzeugen wie Visual Studio, .NET Roslyn und Wisej.NET werden spezialisierte kommerzielle Werkzeuge eingesetzt, etwa:

- TestComplete/SmartBear für automatisierte Tests
- Ozcode für erweitertes Debugging
- NDepend zur Prüfung von Abhängigkeiten

Zudem wird eine lange Liste von im eigenen Haus entwickelten Werkzeuge genutzt, vor allem:

- winformPORTER führt ein regelbasiertes Web-Enabling einer WinForms-Anwendung durch
- accessPORTER migriert eine MS-Access-Anwendung nach Web/Wisej.NET
- vbPORTER Ultimate migriert eine VB-6-Anwendung nach C# oder VB.NET
- TAT (Test Assistance Tool) fasst einzelne Screenshots zu einer Sammlung von Bildern zusammen, erlaubt die Weiterverarbeitung und Ergänzung um Kommentare zur Fehlerdokumentation
- codeANALYZER untersucht Sourcecode auf bekannte Problemmuster, die ggf. manuell angepasst werden müssen
- fixLAYOUT erledigt die automatische Positionierung und Anpassung weiterer Eigenschaften von Controls nach entsprechenden Vorgaben
- compareDESIGN ist eine Visual Studio Extension zum Vergleich von Designer-Dateien
- frxEXTRACTOR kümmert sich um das Auslesen von binär abgelegten Eigenschaften in *frx*-Dateien
- propbagEXTRACTOR verschiebt Designer-Properties von Drittherstellter-Controls in eine PropBag-Datei
- convertRESOURCES liest VB-6-*.res*-Dateien aus und speichert den Inhalt
- getVB6CODE öffnet die korrespondierende VB-6-Datei und Codezeile zur aktuell ausgewählten migrierten Codezeile
- formsOPENER erlaubt das Öffnen beliebiger Form Windows zur Laufzeitüberprüfung beim UI-Redesign aus der Anwendung heraus, ohne dass die Anwendung tatsächlich lauffähig ist
- propertiesINFO zeigt in einem Property Window zur Laufzeit die Eigenschaften des aktuell gewählten Controls

Wenn es um die Architektur geht, gibt es eigentlich nur einen Herkunftsweg: Praktisch alle vorhandenen Pakete auf dem Markt sind bislang monolithisch aufgebaut und folgen einer *stateful*-Architektur. Der gängigen Annahme, dass solche Anwendungen nicht skalieren und sich deshalb grundsätzlich nicht für den Webeinsatz eignen, muss jedoch widersprochen werden. Eine Geschäftssoftware, die an einzelne Kunden verkauft wird, hat pro Kundeninstallation relativ wenige Anwender:innen – einige tausend können schon als große Installation gelten. Für eine solche Zahl reicht eine *stateful*-Architektur allemal aus, sodass jede monolithische Geschäftsanwendung automatisiert in eine Webanwendung transformiert werden kann – mit voller Plattformunabhängigkeit und ohne das Team zu zwingen, sich mit der Komplexität von Microservices und Ähnlichem zu beschäftigen.

Eine bewährte Methodik

Das Unternehmen des Autors bietet seit Jahren als Dienstleistung eine toolgestützte Migration von Altanwendungen nach .NET mit und ohne Web-Enabling an. Im Laufe von rund 500 Projekten hat sich ein standardisiertes Vorgehensmodell herausgebildet und bewährt, das im Folgenden beschrieben werden soll.

Den Start jedes Modernisierungsprojekts bildet eine Evaluierungsphase, die sehr schnell durchgeführt werden kann und zunächst auf rein statistischen Daten der Software basiert. Gesammelt werden diese von einem Evaluierungswerkzeug, das der potenzielle Kunde über seinen Code laufen lässt. Hieraus lassen sich die Projektkosten bereits auf 20 Prozent genau schätzen. Den nächsten Schritt bildet ein Proof of Concept, der die Kosten konkretisiert und dem Auftraggeber ein prototypisches Ergebnis für die fertige Migration liefert. In dieser Phase erfolgt auch die Auswahl der Controls, die für das spätere Projekt zum Einsatz kommen sollen. Eine wichtige Stolperfalle in diesem Zusammenhang wäre, zunächst nur eine partielle Analyse durchzuführen, statt den gesamten Code zu analysieren. So entsteht eine falsche Kostenschätzung, da der Projektpreis sich nicht linear zum Codeumfang verhält. Ebenso gilt es für den Proof of Concept einen repräsentativen Ausschnitt aus der Anwendung zu wählen, damit mögliche Knackpunkte für das Projekt schon in dieser Phase entdeckt und beim Festpreisangebot berücksichtigt werden.

Die eigentliche Migration erfolgt iterativ unter Einsatz verschiedenster, größtenteils im eigenen Haus entwickel-

ter Modernisierungswerkzeuge, deren Auswahl sich nach der Quell- und Zielplattform richtet – beispielsweise dem winformPORTER (**Abb. 4**) – und die zudem durch passende Basisklassen und Foundation Layer vorbereitet sind. Die Quelltexte für die Zielsprache werden grundsätzlich statisch generiert, hinterher aber mit den Mitteln des .NET-Compilers Roslyn dynamisch bearbeitet. Eine wesentliche Stolperfalle bei der Migration ist der erforderliche Code-Freeze: Bei großen Projekten muss der Quellcode über eine gewisse Zeit eingefroren werden. Geht man andererseits nur mit kleinen Teilen modular voran, kann ein Fehlerbild eben nur modular und nicht in der Gesamtheit des Quellcodes behoben werden. Damit sinkt die Produktivität und es entstehen leicht Inkonsistenzen. Weitere Stolperfallen können Ressourcen in Controls darstellen. So verbergen sich in den bislang eingesetzten Controls häufig Konfigurationen und Code, die nicht im Quellcode des Projekts abgelegt sind, sondern in speziellen binären Dateien des Herstellers. Hier sind große Erfahrung und spezialisierte Werkzeuge nötig, um solche Informationen Control-Hersteller-spezifisch auszulesen und passend für die neu eingesetzten Controls zu transformieren.

Abb. 4: winformPORTER

In der nächsten Phase geht es um Test und Finalisierung. Diese erfolgen explorativ und User-Interface-basiert, zunächst technisch, dann anhand von Screen-Videos der alten Anwendungsfunktionalität durch Mitarbeiter des Dienstleisters und in der finalen Phase schließlich durch den Auftraggeber. Insbesondere bei Web-Enabling-Projekten kommen Performance-, Last- und Memory-Tests hinzu, um zu vermeiden, dass solche Probleme erst später im Echtbetrieb auftauchen. Grundsätzlich gilt die Regel „find one, fix all“: Ein gefundenes Problem wird also nicht isoliert gefixt, sondern eine Regel für eins der Tools geschrieben. So wird das Problem an allen Stellen behoben, an denen die entsprechende Konstruktion im Code vorkommt. Größte Stolperfalle in dieser Phase ist, dass oftmals nicht genügend viele oder repräsentative Testfälle für die Masken zur Verfügung stehen oder die Datenbasis nicht ausreichend mit Beispieldaten befüllt ist. Auch alte Fehler, die schon lange unbemerkt in der Legacy-Anwendung vorhanden waren, können zur Stolperfalle werden – etwa ein Memory Leak, das erst im 24/7-Betrieb relevant wird. Zudem ist in dieser Phase ein gutes Monitoring der Tests wichtig, um zu wissen, welcher Teil des Quellcodes bereits abgearbeitet wurde (Code Coverage).

Ist die eigentliche Migration abgeschlossen und getestet, kann es ans User-Interface- oder User--Experience-Redesign gehen. Auch das lässt sich ein gutes Stück weit automatisieren, etwa für ein Facelift durch eine toolbasierte Neuanordnung von Controls. Sogar eine Reduzierung der Anzahl von Controls ist möglich, indem die Werkzeuge. etwa Label, Datenfeld und Button, die in der alten Plattform separate Elemente waren, zu einem kombinierten .NET Control zusammengefasst werden. So muss beispielsweise für die Responsiveness nur ein Control behandelt werden und nicht drei. Hier lohnt sich auch der Einsatz von Profilen, die eine Maske unterschiedlich aussehen lassen, je nachdem, ob sie vom Desktop oder Laptop, vom Tablet oder Smartphone

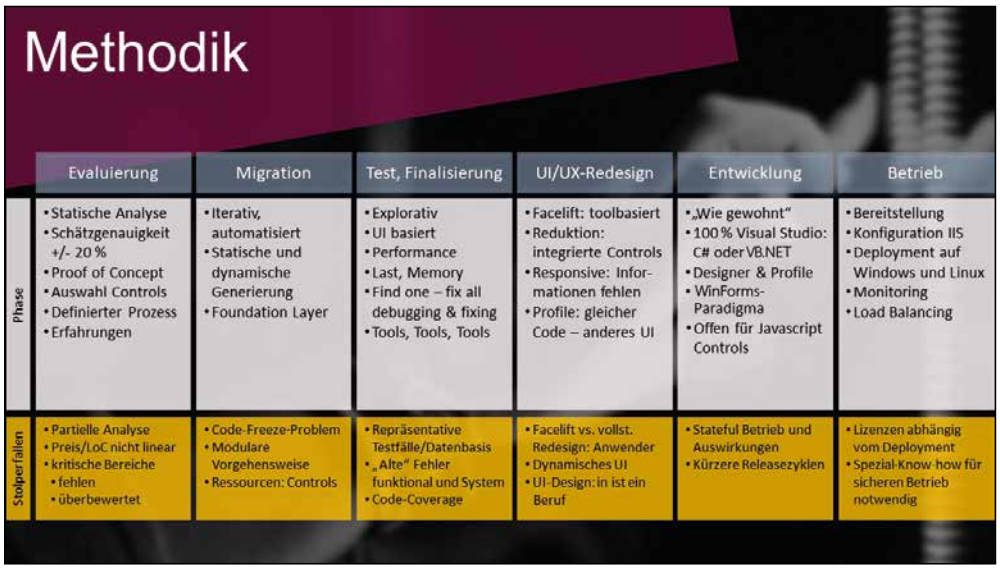


Abb. 5: Methodik der Anwendungsmodernisierung

aufgerufen wird. Eine gängige Stolperfalle in dieser Phase ist es, zu glauben, Entwickler wären auch per se gute Designer. Um das Design des User Interfaces sollten sich unbedingt Personen kümmern, die das professionell beherrschen. Ebenfalls häufig unterschätzt wird der dynamische Einfluss von Quellcode auf das Maskendesign. Es genügt eben nicht, eine Maske in Visual Studio zurechtzuschieben, ohne dabei den Code zu verstehen, der dynamisch das User Interface verändert. Dazu braucht man Werkzeuge, mit denen Entwickler:innen tatsächlich das dynamische UI sehen, ohne immer wieder neue Daten eingeben zu müssen. Und schließlich muss man sich grundsätzliche Gedanken um das bezweckte Ergebnis machen. Reicht ein Facelift aus, bei dem der identische Inhalt der Masken mit etwas neuen Farben und Formen frisch angeordnet wird oder ist ein komplettes Redesign notwendig, bei dem ganz neue Masken entstehen, die durch eine andere Navigation erreicht werden sollen?

Ist das Projekt abgeschlossen, erfolgt die weitere Entwicklung mehr oder weniger wie gewohnt. Der Einsatz des Wisej.NET-Frameworks zum Web-Enabling macht es möglich, dass die weitere Entwicklung komplett in Visual Studio erfolgen kann. Als Sprache stehen C# oder auch VB.NET zur Verfügung, ebenso ein Designer mit Profilen, und es wird nach dem gewohnten WinForms-Paradigma gearbeitet. Zusätzlich können jedoch beliebige JavaScript-Controls eingebunden werden, was die zukünftigen Möglichkeiten stark erweitert. Zu den Fallstricken gehört der *stateful*-Betrieb mit seinen Auswirkungen: Über kurz oder lang wird ein Administrator benötigt, der eine Webapplikation konfigurieren kann und das Thema Load Balancing umsetzt. Außerdem werden bei einer Webanwendung die Releasezyklen kürzer. Statt ein oder zwei Mal im Jahr ein Update auszuliefern, gibt es in Zukunft vielleicht monatliche oder noch häufigere Aktualisierungen, was mittelfristig die Art der Entwicklung und Planung verändern wird.

Damit sind wir bei der Frage nach dem langfristigen Betrieb der Webapplikation angekommen. Waren die Serverwelten von Windows und Linux bislang kaum vereinbar, gibt es ab .NET 7 auch die Möglichkeit, auf Linux zu deployen. Je nach Plattformwahl werden geeignete Verfahren zum Monitoring und zum Load Balancing benötigt. Eine Stolperfalle ist das Thema Lizenzen. Einerseits muss man die Lizenzkosten eines Windows-System versus Linux betrachten, aber auch Lizenzunterschiede bezogen auf das Deployment, gerade in der Cloud. Ob man nämlich mit Infrastructure as a Service quasi einen eigenen Server in der Cloud oder nur einen App-Service bucht, macht einen erheblichen Unterschied – sowohl für die Kosten als auch die damit zu bedienenden Kunden- und Nutzerzahlen. Hier ist ebenso spezielles Know-how notwendig wie zur Herstellung eines sicheren Betriebs.

Am Ende zählen Resultate

Im Lauf von mehr als 10 Jahren haben sich für den Autor diese Methodik (**Abb. 5**) und die entwickelten Werkzeuge (Kasten: „Für die Anwendungsmodernisierung genutzte Werkzeuge“) für die Anwendungsmodernisierung in Auftragsprojekten unterschiedlichster Größe bewährt. Der toolbasierte Ansatz konnte gerade bei komplexen Anwendungslösungen punkten:

- Für ein kommerzielles ERP für die Verpackungsindustrie mit 2,5 Mio. Codezeilen, 115 Anwendungen und 2 100 Masken lag der geschätzte Aufwand zur Neuentwicklung bei 32 000 Personentagen. Die toolbasierte Migration von Gupta nach Web/Wisej.NET benötigte 2 500 Personentage.
- Eine Verwaltungssoftware für Städte und Kommunen mit 1,5 Mio. Codezeilen, 20 Anwendungen und 1 500 Masken. Hier waren für die Neuentwicklung 24 000 Personentage angesetzt, die toolbasierte Migration war nach 2 300 Personentagen abgeschlossen.
- Ein ERP-System für den Großhandel mit 950 000 Codezeilen, 400 Anwendungen und 1 600 Masken sollte eigentlich in 5 800 Personentagen ein Refactoring für das Web erfahren. Die toolbasierte Migration erledigte die Umstellung in 600 Personentagen.
- Der Hersteller einer Branchenlösung für die Wohnungswirtschaft mit 2,95 Mio. Codezeilen und 266 Anwendungen mit 1 400 Masken hatte für das Web-Refactoring 18 000 Personentage kalkuliert. Die toolbasierte Umstellung benötigte 1 500 Personentage.

Mittlerweile sind die Erfahrungswerte so fundiert, dass die Modernisierungsprojekte zum Festpreis und mit garantiertem Fertigstellungstermin angeboten werden und somit weitgehend risikofrei für die Auftraggeber sind. Wer die genannten Fallstricke beachtet, kann jedoch auch bei der eigenen Anwendungsmodernisierung die größten Risiken umgehen.

Als Geschäftsführer von fecher ist **Günter Hofmann** für die Unternehmensleitung und -strategie verantwortlich. Während seiner mehr als 20 Jahre im Unternehmen begleitete er bereits hunderte Software- und Modernisierungsprojekte.

Links & Literatur

[1] https://boomi.com/content/ebook/vanson-bourne-report/

[2] https://www.modernizing-applications.de

[3] https://www.modernizing-applications.de/ihre-informationen/referenzen/

[4] https://www.modernizing-applications.de/ihre-informationen/aktuelles/webinare-vom-november-2023/